# Bulk File I/O Extensions to Java

Dan Bonachea
EECS Department
University of California, Berkeley
Berkeley, CA 94720, USA
bonachea@cs.berkeley.edu

## ABSTRACT

The file I/O classes present in Java have proven too inefficient to meet the demands of high-performance applications that perform large amounts of I/O. The inefficiencies stem primarily from the library interface which requires programs to read arrays a single element at a time. We present two extensions to the Java I/O libraries which alleviate this problem. The first adds bulk (array) I/O operations to the existing libraries, removing much of the overhead currently associated with array I/O. The second is a new library that adds direct support for asynchronous I/O to enable masking I/O latency with overlapped computation. The extensions were implemented in Titanium, a high-performance, parallel dialect of Java. We present experimental results that compare the performance of the extensions with the existing I/O libraries on a simple, external merge sort application. The results demonstrate that our extensions deliver vastly superior I/O performance for this array-based application.

## Keywords

Bulk, Asynchronous, Java, I/O

## 1. INTRODUCTION

One of the defining characteristics of high-performance scientific applications is that they frequently involve massive amounts of I/O. For example, Rosario [20] reports that typical modern supercomputer applications involve anywhere from 1GB to 4TB of I/O per run and I/O rates of up to 40MB/s. The primary techniques for maintaining high performance with this level of disk activity are tuning the data distribution across nodes and providing implicitly or explicitly overlapped I/O. Data distribution seeks to balance the disk workload across nodes and is an important and well-studied topic ([1], [12], [19]), but is orthogonal to the scope of this paper. This research focuses on maximizing the I/O performance for each node given a fixed workload.



Originally published in the Proceedings of the Java Grande 2000 Conference, San Francisco, California USA (June 3-4, 2000)

Unfortunately, the interface[1] to the existing Java I/O libraries is ill suited for the I/O requirements of high-performance, array-based applications. The methods which provide file I/O limit an application to transferring data a single value at a time, which implies a method call overhead linear in the number of primitive data values. This situation is entirely unacceptable for scientific applications, which frequently perform I/O on large arrays of data.

This research proposes and implements a two-part solution. The first part is a small, straightforward extension to the existing Java I/O libraries that enables bulk I/O. This extension alone provides an enormous improvement in performance for a large class of I/O–bound applications by removing the limitations imposed by the existing library interfaces. The second part is a new library that provides direct support for asynchronous bulk I/O, making it easier to mask disk latency with overlapped computation. This interface can provide an additional performance boost to applications with a heavy, yet predictable I/O workload.

The extensions were implemented and tested in Titanium [26], a high-performance, parallel dialect of Java. The extensions utilize some native methods (whose implementation is specific to the Titanium runtime system) but the library interfaces are appropriate for standard Java and the implementation could easily be ported. It is our belief that the concepts explored and performance results are relevant to any high-performance dialect of Java that wishes to support the I/O demands of array-based, data-intensive applications.

Section 2 provides background information about the existing I/O primitives in Java, the Titanium language, and related work. Section 3 presents the bulk synchronous extensions. Section 4 presents the bulk asynchronous extensions. Section 5 presents performance results. Section 6 describes limitations and future work, and we conclude in Section 7.

## 2. BACKGROUND

### 2.1 Bulk I/O in Java

The lack of support for bulk I/O in Java is a well-known design deficiency that has caused problems in adapting Java for use in high-performance computing. Dickens and Thakur [6] present a detailed investigation of this problem and we shall summarize the issues here. Because Titanium is a superset of Java 1.0, we have restricted our attention to the I/O primitives present in that

[1] Throughout this paper, the word "interface" is used in the sense of the Application Programming Interface (API) of the class under consideration, not in the sense of a Java interface, which is an object-oriented language construct. Also, the phrase "I/O" is used to mean "file I/O".

version of the specification; however, Dickens and Thakur report the situation remains essentially unchanged in newer flavors of Java.

File I/O in Java comes in two basic forms. The java.io.RandomAccessFile class provides unbuffered random file access using a seekable file-pointer similar to the fread()/fwrite() functions in the C runtime library. The java.io.FileInputStream and java.io.FileOutputStream classes act as a bottom-level stream interface for sequential, unbuffered, stream-based file I/O. The java.io library provides a rich set of stream operators which can be composed to provide various filtering and buffering characteristics. For example, a common composition for input is:

```
DataInputStream dis = new DataInputStream(
                  new BufferedInputStream(
                  new FileInputStream(<filename>)))
```

where the BufferedInputStream provides buffering, and the DataInputStream performs translation from the abstract stream to useful Java primitive data types. Both interfaces are rather elegant and make for writing clear, well-abstracted code – however, they are both poorly suited to meet the performance demands of scientific or data-intensive applications that involve large amounts of file I/O.

The primary problem shared by both interfaces is they provide essentially no support for bulk I/O transfers between the library and the application variables. All accesses must go through the read*() and write*() methods of the DataInput and DataOutput abstract interfaces, which only allow reading/writing a single primitive data value at a time (e.g. readLong(), writeDouble(), etc.) The only bulk operations provided are for byte arrays, which are used to implement the stream interface. However, the type safety of Java prevents one from efficiently doing much useful work with these byte arrays; at best one can parse the values one at a time (by using them to construct a stream or String, or in a platform-dependent way using arithmetic).

As a consequence of this interface, file I/O in Java commonly involves tight loops that transfer file data to or from an array where it is worked on – this is especially true for scientific or data-mining applications that frequently employ large array data structures. Needless to say, this implies significant software overheads associated with array I/O, as the number of method calls is linear in the number of primitive input/output values, rather than in the number of arrays the application wants to read or write[2]. This is a significant oversight on the part of the Java designers, and limiting the interface in this manner creates enormous I/O bottlenecks in applications that perform large amounts of file I/O. The first extension implemented in this project solves this problem by adding bulk I/O primitives to the libraries that perform I/O on entire arrays with a single method call.

[2] To make matters worse, the composition of streams often implies a chain of method call invocations for each primitive input/output call. Furthermore, the implementation of the I/O primitives for multi-byte values sometimes involves multiple calls to the single-byte primitives (e.g. DataInputStream.readLong() is often implemented as 8 calls to the DataInput.read() method that inputs a single byte).

It is important to note that the semantics for both existing interfaces are entirely synchronous. The BufferedInputStream / BufferedOutputStream classes provide implicit prefetching and write-behind caching in a private buffer (of a user-specified size), but all buffer reads and writes block the thread of control issuing the request that triggers a buffer flush/refill. There may be some opportunity for adding asynchronous buffer management at this level of the abstraction; however, because users are free to compose streams in any way they wish and even define streams of their own, this could produce unpredictable and often undesirable results (especially when the bottom level is something other than a file, such as a communication pipe to a different process). A more subtle difficulty with the stream buffering classes, recognized by Heydon and Najork, is that they impose a non-trivial synchronization overhead in order to maintain thread-safety [9].

## 2.2 Titanium

Titanium is a high-performance, explicitly parallel, SPMD dialect of Java developed at U.C. Berkeley for programming shared-memory and distributed-memory parallel systems. Titanium incorporates the power of Split-C [25], a low-level SPMD language, into a high-level object-oriented programming language that frees the programmer from much of the tedium associated with writing and debugging parallel programs. Titanium is essentially a superset of Java 1.0, including all the expressiveness and safety features of that language, with a wealth of new features that support high-performance SPMD programming, such as: user-defined immutable classes, zone-based memory management, local and global references, flexible and efficient multi-dimensional arrays, unordered loop iteration, and a library of useful parallel primitives including barrier, broadcast, exchange, and various reductions [2], [29], [10]. The compiler performs extensive static analysis (with some assistance from programmer-inserted type qualifiers) to statically guarantee freedom from deadlock on barrier synchronization [7]. The primary goals of the language, in order of importance are: performance, safety and expressiveness. Titanium is especially well adapted for writing grid-based scientific parallel applications, and several such major applications have been written and continue to be further developed [25].

The Titanium compiler performs various optimizations using knowledge of the parallel control flow, and translates programs entirely to C, where they are compiled (and optimized further) by a C compiler and then linked to the proper Titanium runtime libraries (there is no JVM). The Titanium backend has been ported to several platforms, including SMP's running Solaris or Posix threads, Solaris and Linux uniprocessors, Cray T3E, IBM SP2, Tera MTA and the Berkeley NOW (a cluster of UltraSPARC's [5], [17]).

The only major Java feature which is not currently supported by Titanium is arbitrary multi-threading. That is, the user cannot create arbitrary asynchronous threads of control within a process. Although Titanium does not employ an explicit message-passing paradigm, the SPMD control model is central to the design of the language, and this programming paradigm does not typically accommodate arbitrary threading [3]. A Titanium program runs on a fixed set of processors (which may be virtual processors in the case of uniprocessors, or real processors in the case of SMP's and distributed-memory multiprocessors) – the number of proc-

essors is determined when the parallel job begins, and remains constant throughout the run of the program. This property is useful in implementing the optimizations and features of Titanium, and crucial to the static prevention of deadlock.

One significant drawback to Titanium's restriction on arbitrary multi-threading is that applications cannot implement asynchronous file I/O using the blocking I/O primitives from the Java libraries. Consequently, Titanium applications with demanding I/O requirements have no way to hide disk latency with overlapped computation. The AsyncFile library created by this project alleviates this problem by providing the Titanium programmer easy access to efficient, non-blocking I/O routines that permit I/O to be overlapped with continued computation and network activity[3].

## 2.3 Asynchronous File I/O

Historically, asynchronous I/O has often been cited as an important optimization in masking disk latency, but it is rarely implemented at the level of the application or even the runtime system, and there is no widely accepted standard interface. The reason seems to be that the implementation of synchronization and threading varies considerably across languages and operating systems. Most modern operating systems implement some form of file-system buffering (which can be seen as a limited form of asynchronous I/O) that effectively overlaps some disk latency with ongoing computation. However, this buffering tends to work best with sequential reads and writes (where sequential prefetching and write-behind caching help out), and cannot remove the latency and resource overhead associated with copying the data to and from the OS buffers [11], [23], [28].

### *2.3.1 Overview of Asynchronous I/O*

Implementations of asynchronous I/O cluster around two major points in the design space. The first category is multi-threading of synchronous I/O – where the application programmer writes code to explicitly create a new thread that calls a standard blocking I/O interface, and to synchronize that thread with the computation thread once the I/O completes. This approach places the greatest burden on the application and can quickly lead to very messy code or deadlocks because the programmer has to manage the I/O synchronization explicitly. This approach does have the advantage of being the more portable of the two – although there are still difficulties because the multi-threading interface can vary somewhat between platforms. This approach also seems to work best when programmed to be explicitly aware of the virtual memory system – first, to prevent buffers from being swapped out to disk, and second, to reduce the number of intermediate in-memory data copies to one or zero. This tends to make implementations even more platform-specific.

The second option relegates the bulk of the complexity to the kernel or an I/O library, which manages the various I/O threads internally, providing the programmer with a relatively simple interface for making requests and retrieving results. There are variations in how these implementations communicate results to the application; some use an interrupt-based approach, where the application receives a signal or a call to a handler routine when the operation completes. Others use explicit synchronization, where the application must poll the status of or wait for the results of an ongoing asynchronous I/O operation. Still others are some combination of these approaches. All such interfaces decree that the application must not touch the memory buffer associated with the request throughout the duration of the operation; this property allows the library to service the I/O request directly on the memory buffer without making an additional copy to prevent race conditions.

### *2.3.2 Brief Case Study of Asynchronous I/O*

One of the major challenges in this project was designing a flexible, concise and efficient interface for asynchronous I/O in Java. The design of the AsyncFile library was heavily influenced by several existing asynchronous I/O interfaces, which we briefly outline here.

The Microsoft Windows NT operating system provides direct support for "Overlapped Files" within the Win32 API [15]. In this interface, the application initiates an I/O operation and specifies either a handle for the operation or a completion routine callback. The application can check the status of ongoing operations, stop and wait for a particular operation to complete, or poll for any queued completion routines (callback routines execute synchronously when the application explicitly indicates it is ready to receive them). The interface allows the application to specify at file open time whether the I/O should be buffered (which implies an additional copy to/from the OS buffers) or unbuffered (zero copies – the buffer passed by the application is used directly by the low-level disk I/O driver). The important caveat associated with unbuffered files is that all I/O must be made in sizes that are an even multiple of the disk volume's sector size, and the I/O requests must be sector aligned in memory and on-disk. This is a significant limitation, but is fundamental to any zero-copy scheme because all modern disks perform I/O in aligned sector units [21], [22]. The advantage to such a low-level interface is that the application can directly control its caching behavior using application-specific knowledge; this technique is especially effective in applications such as database management systems that are designed from the ground up to exploit this level of control.

The Solaris operating system provides primitive support for buffered asynchronous I/O through the aio library [24]. The functions aioread() and aiowrite() are used to initiate operations, and synchronization is achieved by catching the SIGAIO UNIX signal sent on completion or calling the aiowait() function to poll or wait for an operation to finish. Note that unlike other synchronization implementations, the aiowait() interface is not request specific – that is, it waits for any operation spawned by the current process to complete without allowing the programmer to specify which requests are of interest. This is an inconvenience that was remedied in the new Solaris Posix4 library, which offers request-specific polling and waiting, a more flexible signaling interface, application-directed request prioritization, and support for asynchronous buffer flushing requests. Unfortunately, this new interface is only available for Solaris 2.6 and higher. The prototype discussed in this paper was implemented using the basic aio library in the interests of maximizing portability, but

[3] Incidentally, asynchronous I/O can also make applications more tolerant to irregularities in disk performance due to contention.

future incarnations may move to the newer interface in order to simplify the control logic.

The Message Passing Interface (MPI) Standard, an influential standard in the parallel computing community, has included buffered asynchronous I/O primitives in the new MPI 2.0 Specification [14] through the MPI_FILE_IREAD_* and MPI_FILE_IWRITE_* initiation functions, using the MPI_TEST and MPI_WAIT functions for synchronization. Implementations of the specification are still free to use blocking semantics for these calls if the hardware is unsuitable, but the fact they were included in the specification is a clear sign that OS-managed, asynchronous I/O has been recognized as an important point in the design space for high-performance computing. Hopefully this will prompt more OS vendors to incorporate such primitives into their system call or standard library interfaces.

# 3. BULK SYNCHRONOUS EXTENSIONS

## 3.1 Library Interface

Figure 1 presents the public declarations for our bulk synchronous extensions to the I/O libraries. The extensions use subclassing to add support for bulk I/O to the java.io.* classes that implement the DataInput/DataOutput abstract interfaces [8]. The new bulk I/O methods are readArray() and writeArray(), which take a single-dimensional array of primitive type that will participate in the I/O. Overloaded forms of these methods take additional arguments (array offset and element count), allowing the programmer to specify a contiguous subset of the elements to be operated on. The array argument is declared as having type Object to succinctly accommodate all of the permitted array types; in Titanium, this also includes arrays of atomic immutable objects. The classes do not directly support arrays of multiple dimensions or whose elements are of reference type, as discussed in section 6. Given these methods, the programmer can perform bulk I/O on arrays of arbitrary length with a constant method call overhead.

```
package ti.io;
public class BulkRandomAccessFile extends RandomAccessFile {
  public void readArray(Object array) throws IOException;
  public void writeArray(Object array) throws IOException;
  public void readArray(Object array,
     int arrayoffset, int count) throws IOException;
  public void writeArray(Object array,
     int arrayoffset, int count) throws IOException;
}
public class BulkDataInputStream extends DataInputStream {
  public void readArray(Object array) throws IOException;
  public void readArray(Object array,
     int arrayoffset, int count) throws IOException;
}
public class BulkDataOutputStream extends DataOutputStream{
  public void writeArray(Object array) throws IOException;
  public void writeArray(Object array,
     int arrayoffset, int count) throws IOException;
}
```

**Figure 1: Bulk Synchronous I/O Extensions**

## 3.2 Implementation

The methods are implemented natively by casting the array argument to a byte array and passing it to the byte array I/O methods of the parent classes. Leveraging the existing functionality of the parent classes in this way made the implementation relatively trivial and portable. Note this implementation strategy is only valid because of our restriction to single-dimensional arrays of non-reference type; this restriction guarantees that the array element data all resides contiguously in memory.

The only implementation complexity that arises is maintaining the platform-independent on-disk representation required by the Java standard. Specifically, little-endian implementations of these methods must perform a byte-swapping pass on the array data, in order to ensure that data is written out in big-endian order as required by the Java standard [8].

## 3.3 Safety Issues

The new I/O primitives maintain the level of language safety present in the legacy Java I/O library. A rigorous proof is beyond the scope of this paper, but we sketch the reasoning here. Intuitively, the bulk methods accomplish what can already be done given the Java I/O primitives, albeit much faster. Using an appropriate composition of DataOutputStream and ByteArrayOutputStream objects, it is possible to change an arbitrary list of Java primitive values into a single dimensional, untyped byte array using the write*() methods and a loop. Similarly, DataInputStream and ByteArrayInputStream allow one to extract an arbitrary list of Java primitive values from an untyped byte array. These untyped byte arrays can be used to perform bulk I/O using the existing methods in the DataInput/DataOutput interface (these only provide bulk I/O for byte arrays).

The bulk extensions accomplish exactly this behavior[4], except they do it much faster by reducing the number of method calls necessary to a small constant, providing enormous speed-ups in practice. We suspect that the observed speedups may be even more significant on JVM-based Java dialects where the native method call overhead is even more significant than in Titanium (where all method calls cost the same as a C function call).

# 4. BULK ASYNCHRONOUS EXTENSIONS

## 4.1 Library Interface

The second extension proposed and implemented in this research is the AsyncFile class. The goals of the library, in order of importance, are performance, power, and simplicity. Performance issues are addressed by providing support for asynchronous bulk file I/O, allowing I/O to proceed with overlapped computation. Power, that is to say expressiveness, was achieved by studying the existing interfaces for asynchronous I/O and formulating a design that takes the best features of each and places them in a Java framework. Specifically, the synchronization primitives were designed to be very flexible and expressive. Lastly, simplicity was achieved by adhering to the Java "flavor" wherever possible and minimizing the number method calls required to perform an I/O operation, thereby making the interface easy to learn and use.

[4] There is actually a very subtle difference that may arise depending on how the bulk extensions are implemented. If the "casting" operation is implemented as a literal type-cast, then the byte array produced will be an alias of the typed array. Implementations in safety-critical dialects can allocate a temporary buffer and perform a single memcpy() operation to remedy this detail, however our implementation chooses to ignore it in the interests of maximizing performance.

### 4.1.1 File-level Control

Figures 2 and 3 present the public declarations of the AsyncFile and AsyncFileRequest classes, which constitute the interface of new library. We sketch the semantics of the library below; the detailed specifications are available on the Titanium web site [25]. The interface is similar to the RandomAccessFile class, in that the user is given the ability to seek to random positions in the file, although the default behavior is to service requests sequentially. The AsyncFile class is the abstraction for a non-blocking, buffered[5] file with support for bulk I/O. The methods for file-level control are self-explanatory and identical to those provided by RandomAccessFile [8].

```
package ti.io;
public class AsyncFile {
  // -------  File-level Control ---------
  public AsyncFile(String filename, String mode)
     throws IOException, IllegalArgumentException;
  public AsyncFile(File filename, String mode)
     throws IOException, IllegalArgumentException;

  public long length() throws IOException;
  public long getFilePointer() throws FileNotOpenException;
  public void seek(long offset) throws IOException;
  public void close() throws FileNotOpenException;
  public void cancel() throws IOException;

  // ------- Request Initiation ---------
  public AsyncFileRequest readArray(Object array)
     throws IOException;
  public AsyncFileRequest writeArray(Object array)
     throws IOException;
  public AsyncFileRequest readArray(Object array,
     int arrayoffset, int count) throws IOException;
  public AsyncFileRequest writeArray(Object array,
     int arrayoffset, int count) throws IOException;

  // ------- Request Synchronization ---------
  public static final int TIME_INF;
  public boolean allDone(int timetowait) throws IOException;
  public static boolean anyDone(int timetowait,
      AsyncFileRequest [] requestlist) throws IOException;
  public static boolean allDone(int timetowait,
      AsyncFileRequest [] requestlist) throws IOException;
}
```

**Figure 2: AsyncFile Class**

### 4.1.2 Request Initiation

The second section of Figure 2 documents the I/O initiation methods. The I/O initiation methods are readArray() and writeArray(), which work similarly to the analogous methods in the bulk synchronous classes. When these methods are called they perform type-checking, bounds-checking and end-of-file-checking, then initiate the requested asynchronous I/O operation. These methods return an AsyncFileRequest object (see Figure 3), which serves as the application's handle to the non-blocking operation that was initiated. This handle is used in subsequent calls to perform synchronization.

Once the asynchronous request has been successfully initiated, control is returned to the application. While the non-blocking I/O completes in the background the application may perform other arbitrary computations - however, it must not access the array being used for I/O until the operation completes. The application is also permitted to initiate other synchronous or asynchronous file operations to run concurrently.

[5] There is no explicit buffering in the prototype implementation, but it is provided within the OS.

```
package ti.io;
public class AsyncFileRequest {
  public final AsyncFile parent;

  public boolean done(int timetowait) throws IOException;
  public boolean done() throws IOException;

  public void cancel() throws IOException;
  public void clearException();
}
```

**Figure 3: AsyncFileRequest Class**

### 4.1.3 Request Synchronization

The synchronization methods of AsyncFile were designed to be flexible and expressive. Besides the basic capabilities of polling the status of a particular request and waiting for a request completion, the application may also concisely poll or wait for all the requests in-flight for a particular file, or even a set of requests from different files. Some applications with simple requirements may not even need to explicitly keep track of their AsyncFileRequest handles, relying instead on file-wide synchronization.

When the application wishes to synchronize with the ongoing operations, it calls one of the "Done" methods in the AsyncFile or AsyncFileRequest classes. The Done methods each take a time-out parameter specifying an interval in milliseconds that ranges anywhere from zero (poll) to infinity (a specially provided constant that means block indefinitely). The semantics of the Done methods are to check the status of the AsyncFileRequests the application is querying, raise any I/O exceptions that may have occurred, and return a boolean indicating whether the Done condition was satisfied. For example, the AsyncFile method allDone(int) waits for all the pending operations on this AsyncFile object to complete or for the timeout to expire (whichever happens first), then returns true or false to indicate which happened. The static allDone(int, AsyncFileRequest[]) method is similar, except it waits for all the requests in the provided list to complete or for the timeout to expire. Similarly, the static anyDone(int, AsyncFileRequest[]) method returns when at least one of the listed requests is complete (or the timeout expires). The AsyncFileRequest.done(int) method is used for checking just a single request, and is semantically identical to calling AsyncFile.anyDone and passing just this request (the timeout argument defaults to infinity if omitted).

### 4.1.4 Exceptions and I/O Errors

Any of the Done methods may throw an I/O exception if one of the requests being queried has ended in an exceptional condition. In most cases, scientific applications are programmed with fail-abort semantics – that is, I/O exceptions are generally so rare in debugged code that the programmer does not care what happens if one occurs. This default case is handled nicely by the Java exception mechanism; in the absence of catch code, any exception thrown propagates to the top of the program and halts execution with an error. However, support is also provided for applications that wish to catch I/O exceptions and possibly perform some form of recovery - for example, check-pointing their state before shutting down to minimize the cost of lost work.

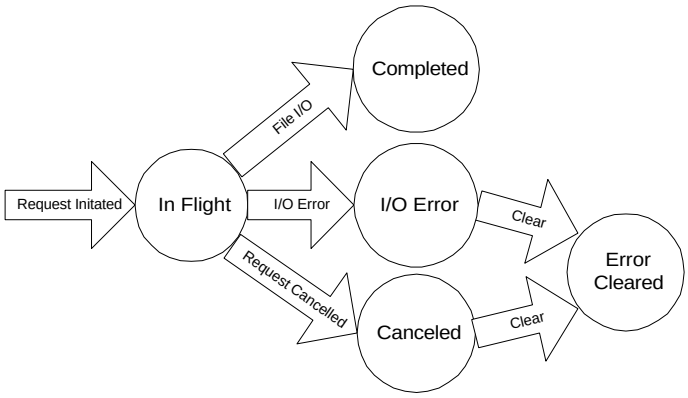


**Figure 4: AsyncFileRequest State Diagram**

Figure 4 illustrates the state diagram for AsyncFileRequest objects. AsyncFileRequest objects begin their life in the "In Flight" state when they are returned from the readArray() and writeArray() methods. The common behavior is for the I/O to complete successfully and for the AsyncFileRequest to move into the "Completed" state where it remains until it is destroyed. However, if an I/O error occurs, the AsyncFileRequest moves into the "I/O Error" state – future calls to the Done methods which include this AsyncFileRequest will cause an exception to be thrown. In the case of an error, the programmer may wish to call the AsyncFileRequest.clearException() method, which moves the exceptional request into the "Error Cleared" state, so that further calls to Done methods which include this request will not rethrow the exception.

The AsyncFileRequest.cancel() method is used to cancel a particular request if it is still in-progress, a handy capability for some applications. This feature may be useful for applications that are reading from slow or highly contended mirrored disks (where a response from either is sufficient, so a request can be issued to both disks and the "loser" can be canceled). It can also be used to support applications that wish to issue speculative prefetches and cancel on misprediction. Canceling a request frees I/O and OS resources, and moves the request into the cancelled state (which is identical to the "I/O Error" state, except the exception thrown will always be an AsyncIOCanceledException).

## 4.2 Asynchronous Library Design Results

We believe the AsyncFile interface meets the design goals of performance, power and simplicity. Performance is provided by enabling overlapped computation and supporting bulk I/O (empirical results are presented in section 5). In terms of power, initial usability results indicate the interface is very expressive and a natural match for the types of synchronization which applications wish to perform. Specifically, the interface makes it very easy to open a file, initiate a number of requests, then work on each of them as they complete, allowing the OS disk scheduler to choose the optimal evaluation order. The random access feature allows applications to jump around to arbitrary parts of a file, and is especially useful for applications where different processors write their results to different assigned areas of a file (for example, in parallel radix sort). The interface also very naturally accommodates double-buffering algorithms, which use a leapfrog approach, performing I/O on one chunk of memory while computing on the other, then switching them. The exception semantics fit cleanly into the Java exception model and default to intelligent behavior. Finally, the implementation fully supports 64-bit file offsets for I/O on massive files.

To test usability and performance, we wrote a benchmark application (Figure 5) that implements the initial sorting pass of an external merge sort. The initial pass of an external merge sort algorithm repeatedly reads a large chunk of a file, performs an in-memory sort, and writes the sorted chunk out to disk. Subsequent passes merge these sorted fragments into larger sorted fragments until only a single sorted fragment remains – however, we have omitted the merge passes to avoid complicating the interpretation of the performance results. The actual computations performed by the benchmark are admittedly somewhat atypical

```
// open files
AsyncFile inf = new AsyncFile("infile.data","r");
AsyncFile outf = new AsyncFile("outfile.data","rw");

// calc the number of chunks reqd. (assumed to be >= 3)
int chunksz = 4096; // 4096 longs = 32 KB chunks
int numchunks = (int)(inf.length()/(chunksz*8));

AsyncFileRequest Req1, Req2, Req3;  // request pointers
long [] B1 = new long[chunksz];     // Allocate buffers
long [] B2 = new long[chunksz];
long [] B3 = new long[chunksz];

inf.readArray(B1).done(); // blocking read(B1)

int chunksleft = numchunks-1;

while (chunksleft > 1) { // the "steady-state" loop

  Req2 = inf.readArray(B2);
  sort(B1);
  Req1 = outf.writeArray(B1);

  // wait( B2, B3 )
  AsyncFileRequest [] tmplist1 = { Req2, Req3 };
  AsyncFile.allDone(AsyncFile.TIME_INF, tmplist1);

  Req3 = inf.readArray(B3);
  sort(B2);
  Req2 = outf.writeArray(B2);

  // wait( B1, B3 )
  AsyncFileRequest [] tmplist2 = { Req1, Req3 };
  AsyncFile.allDone(AsyncFile.TIME_INF, tmplist2);

  // do some swapping
  long [] oldB1 = B1; AsyncFileRequest oldReq1 = Req1;
  long [] oldB2 = B2; AsyncFileRequest oldReq2 = Req2;
  long [] oldB3 = B3; AsyncFileRequest oldReq3 = Req3;
  B1 = oldB3;  Req1 = oldReq3;
  B2 = oldB1;  Req2 = oldReq1;
  B3 = oldB2;  Req3 = oldReq2;

  chunksleft -= 2;
}
if (chunksleft == 1) {
  Req2 = inf.readArray(B2);
  sort(B1);
  Req1 = outf.writeArray(B1);

  Req2.done(); // wait(B2)
  sort(B2);
  Req2 = outf.writeArray(B2);

  // wait( B1, B2 )
  AsyncFileRequest [] tmplist3 = { Req1, Req2 };
  AsyncFile.allDone(AsyncFile.TIME_INF, tmplist3);
}
else { // chunksleft == 0
  sort(B1);
  outf.writeArray(B1).done(); // blocking write(B1)
}
```

**Figure 5: Code fragment from external sort application**

for a scientific application, however the program's behavior is similar to many data-mining applications and has the advantage of simplicity and ease of exposition. Furthermore, the regular pattern of computation and I/O on large chunks of data makes this application a prime candidate for optimization using the AsyncFile library.

The asynchronous version of the application uses double-buffering on the input and output, and cleverly manages to do so using only three memory buffers (this is a useful optimization because the buffers can be made larger while still fitting in physical memory, leading to larger sorted fragments and fewer subsequent merge passes). This optimization is enabled by the fact that multiple I/O operations can be outstanding simultaneously. The relevant piece of the source code is presented in Figure 5 (we have omitted the performance instrumentation code and the body of the sort function, which is a standard O(n log n) quicksort). Although this program has a rather complicated pattern of file I/O, we managed to write the entire I/O skeleton and get it working in less than 30 minutes. We believe this is a major victory in terms of simplicity.

As far as expressiveness is concerned, the double-buffered code that uses AsyncFile occupies about 40 lines of non-comment code, whereas the analogous single-buffered synchronous code written for the Java I/O stream library occupies about 20 lines. When one considers the increased fundamental complexity of the double-buffered algorithm, a 20-line increase seems very modest – we believe this is a testament to the expressiveness of the interface. The code utilizing the asynchronous library adds a few lines of code for request synchronization, but it eliminates value-reading loops, file stream composition, and an EOF exception handler from the code written for the Java I/O stream library.

One subtle feature of the AsyncFile interface is that it concisely allows one to perform synchronous I/O as a special case when no useful overlapped computation can be performed (for example, while reading the first buffer or writing the last buffer in a double-buffered algorithm). One example is the first call to the read() method in Figure 5:

```
inf.readArray(buf1).done(); // blocking read(buf1)
```

When the library is accessed in this synchronous manner, it behaves very similarly to the BulkRandomAccessFile extension (and incidentally, their synchronous performance is nearly identical).

# 5. PERFORMANCE RESULTS

The double-buffered sort presented in Figure 5 which uses the AsyncFile library was re-implemented to use single-buffered, blocking I/O with the new bulk synchronous classes, and also with various configurations of the existing Java I/O libraries in order to evaluate the relative performance gain of the extensions. The I/O library configurations we shall present are listed in Table 1 – the first three are the "legacy" configurations provided by Java, the last three embody the extensions added in this paper. Buffered configurations used the default 2KB memory buffer.

The tests were run using input files that ranged in size from 1 MB to 128 MB, with chunk sizes ranging from 32 KB to 512 KB. We collected a large amount of data, but in the interests of space, we present only a few of the more interesting results. Specifically, chunk size was (not surprisingly) found to have little influence on the total I/O time for the synchronous configurations. All performance data presented in this paper was gathered on a Sun Ultra-1/170 workstation (a node from the Berkeley NOW) running Solaris 5.6 with I/O on the local file system. All tests were compiled using Titanium tc version 1.40 with full optimization and the uniprocessor backend. The relative performance of the various configurations was found to be comparable when using network file systems and other hardware platforms, but these results are not presented here.

Several other configurations were considered and similarly omitted for brevity – for example, adding buffering to the bulkds configuration was found to make no difference in this application. Varying the buffer size on buffered configurations made very little difference over a size of about 128 bytes (2KB is more than sufficient). Using the AsyncFile library with an entirely synchronous algorithm yielded almost the same results as bulkraf (there was a few percent overhead which can be attributed to the bookkeeping code that handles asynchronous I/O initiation and synchronization). We also considered and discarded several buffered and unbuffered configurations using the textual I/O primitives of the Java libraries, which (not surprisingly) always performed significantly worse than their binary counterparts.

## 5.1 Bulk I/O Results

Figure 6 compares the throughput in MB per second for the external merge sort application, ranging over different file sizes on the various configurations. Note that the throughput values graphed in the figures represent the throughput for the entire algorithm, and cannot be directly compared with disk bandwidths for two reasons. First, the data size used to calculate this throughput is the input file size, and the total I/O volume is actually twice that because the algorithm reads and writes every chunk of data that it sorts. Secondly, the time used to calculate this throughput is the total running time, which includes time dedicated to sorting (Section 5.3 presents a detailed breakdown of this running time).

The primary observation to be made from Figure 6 is that the new bulk I/O configurations always beat the legacy Java libraries, by amounts ranging from 2x to over 60x. The slowest configurations were the unbuffered legacy libraries (raf and ds) – the raf configuration is especially significant because it is the only mechanism provided by the legacy Java libraries that allows random file accesses (which are crucial in many applications). Adding buffering to the stream configuration (dsb) gives the best throughput offered by the legacy libraries, but the performance of dsb is still less than half that of the bulk I/O configurations. The bulk synchronous I/O configurations (bulkraf and bulkds) deliver the best synchronous throughput and are roughly similar in performance. Finally, the bulk asynchronous configuration (async) gives a noticeable further improvement over bulkraf and bulkds to achieve the best throughput (making it over 60x faster than raf and ds).

## 5.2 Overlapped Computation Results

In comparing the relative performance merits of bulk synchronous I/O and bulk asynchronous I/O, it is instructive to examine the ideal situation and derive an upper bound for the performance gain possible in the presence of an I/O – computation over-

**Table 1: I/O Configurations Presented**

| name | Description |
|---|---|
| raf | RandomAccessFile() |
| ds | DataInputStream(FileInputStream())<br>DataOutputStream(FileOutputStream()) |
| dsb | DataInputStream(BufferedInputStream(FileInputStream()))<br>DataOutputStream(BufferedOutputStream(FileOutputStream())) |
| bulkraf | BulkRandomAccessFile() |
| bulkds | BulkDataInputStream(FileInputStream())<br>BulkDataOutputStream(FileOutputStream()) |
| async | AsyncFile() - using the double-buffered algorithm of Figure 5 |

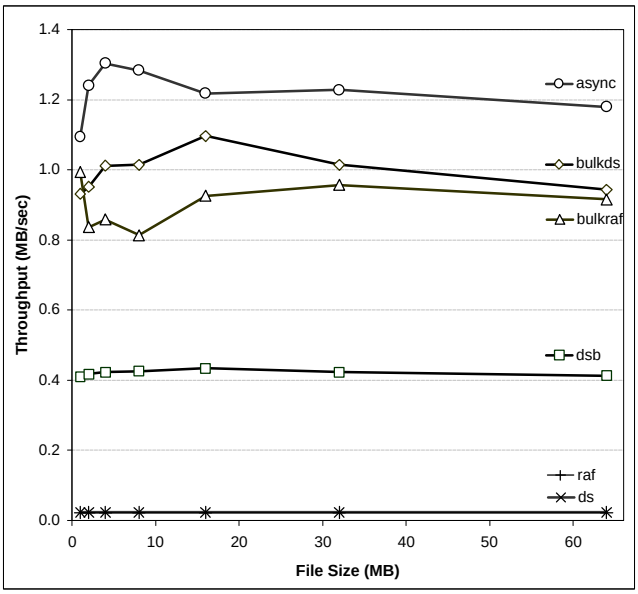


**Figure 6: Sorting throughput with 32KB chunks**

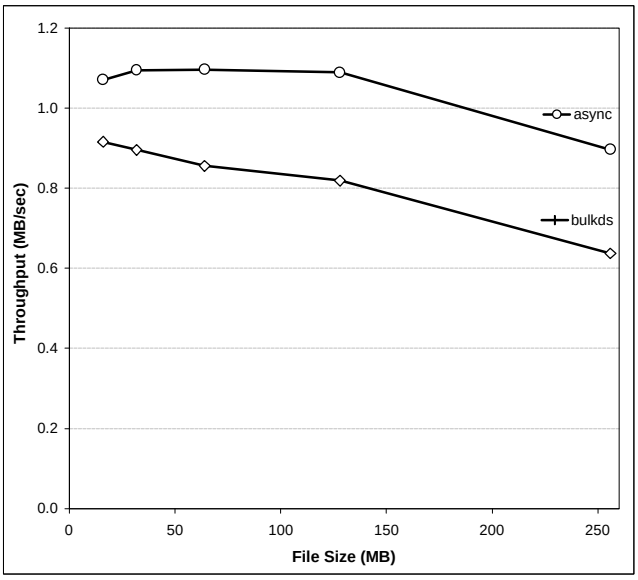


**Figure 7: Sorting throughput with 512 KB chunks**

lap. To this end we start with several overly-optimistic assumptions: 1) the total CPU work and total disk work is identical in either configuration, 2) both resources are completely dedicated to this application, 3) there is no computational overhead for the asynchronous initiation and synchronization routines (they run in zero time), and 4) the computational performance is unaffected by the I/O taking place in the background. The best situation we could hope for is to keep both the CPU and disk busy 100% of the time, and under the above assumptions, this implies a 50% speedup from I/O – computation overlap over the synchronous case where the disk and CPU time is serialized[6].

In practice, several factors stand in the way of reaching the ideal speedup. Asynchronous I/O entails more bookkeeping and startup costs than synchronous I/O - the overhead for the initiation and synchronization routines can be significant. Secondly, CPU performance may be somewhat adversely affected by the I/O taking place in the background, depending on the I/O and memory bus architecture. Finally, there are indirect costs in the application itself that are implied by overlapping I/O with computation – for example, a application designed for overlapped I/O may require additional memory buffers, imposing a larger memory footprint that results in degraded cache performance.

In order to best observe the performance gains of overlapped I/O for this application, it is necessary to balance the CPU and disk workloads by using larger chunk sizes (this increases the total amount of sorting work done by the CPU while the I/O work remains approximately constant). Figure 7 compares the running time performance of async against the fastest synchronous configuration (bulkds) with a chunk size of 512 KB. Because the I/O for the first and last chunks in the async algorithm must still be performed synchronously, the performance gain of async increases as the file size increases and these chunks become negligible with respect to the entire file. As the file size increases to 256 MB, the performance gain of using asynchronous I/O reaches about 28% and tops out at about 30% for larger files. As predicted, practical overheads prevent the async algorithm from achieving the ideal 50% speedup over the synchronous configuration, but a 28% improvement is still quite valuable for I/O-intensive applications which demand the highest possible performance.

## 5.3 Performance Breakdown

Table 2 provides a performance breakdown detailing the areas where the benchmark application spends its time in each of the configurations. The total time is broken down into the time spent reading, writing, sorting and (for the asynchronous algorithm) waiting at synchronization points. The table gives the raw data that corresponds to the 64 MB data points in Figure 6. For comparison, the table also includes the performance breakdown for a bulk synchronous version of the algorithm written entirely in C.

There are many interesting conclusions to be drawn from this breakdown. First, the raf and ds configurations are clearly dominated by I/O method call and system call overheads; the only significant difference between the I/O behavior of ds and dsb is that the buffering in dsb induces far fewer calls to the methods of

[6] This discussion also assumes a storage subsystem that's only capable of servicing a single I/O request at a time. In multi-disk systems that can service several requests in parallel, asynchronous I/O offers the additional benefit of overlapping I/O with other I/O. The performance speedup in such a system is limited only by the total I/O bandwidth and the degree of parallelism in the hardware and application requests.

FileInputStream/FileOutputStream and the underlying read() and write() functions in the library native code. Secondly, the low write times for bulkraf and bulkds clearly reveal that this OS does perform write-behind caching. However, the performance benefits of the write-behind cache are only observed by the bulk I/O configurations where the heavy method call overheads have been reduced to a reasonable level.

**Table 2: Raw Performance Breakdown (in seconds) 64MB file, 32KB chunks (2048 chunks)**

| name | total time | read time | write time | sort time | block time |
|---|---|---|---|---|---|
| raf | 2834.312 | 1298.845 | 1493.879 | 41.538 | |
| ds | 2687.120 | 1229.493 | 1415.939 | 41.577 | |
| dsb | 154.885 | 60.375 | 52.685 | 41.736 | |
| bulkraf | 69.905 | 27.175 | 0.726 | 41.980 | |
| bulkds | 67.828 | 24.943 | 0.874 | 41.925 | |
| async | 54.262 | * 0.500 | * 0.979 | 44.655 | 8.094 |
| C | 39.514 | 24.111 | 0.958 | 14.371 | |

* = these times only include the time to initiate the asynchronous operation

Inspecting the breakdown of the pure C implementation, we discover that the pure I/O performance (read time + write time) of the Java bulk extensions is very competitive with the I/O performance of the C implementation. This is not terribly surprising, because the Java code using the bulk I/O extensions is compiled down to C code that behaves very similarly to the I/O code in the hand-written C implementation. The sorting performance of the Java configurations is less than stellar because the Titanium optimizer is still under development and does not yet extensively optimize the sort function. However, the sort has consistent performance across all the Java configurations, so it can be viewed as a fixed, "black-box" CPU workload - the issue of absolute sorting performance is orthogonal to this investigation.

The breakdown of the async configuration reveals that the performance benefits of this configuration come primarily from the asynchronous reads (because the writes in the synchronous configurations are already performed asynchronously thanks to the write-behind cache). This means it was probably unnecessary to implement write buffering for this particular application and platform – however, application-controlled asynchronous writes are still a useful feature in general. Regardless, the asynchronous reads clearly provide a significant performance win.

The asynchronous breakdown also quantifies the overheads described in Section 5.2. As predicted, the asynchronous request initiation methods have small, but non-zero costs – in this case, they averaged 0.24 ms for each read initiation and 0.48 ms for each write initiation. The slightly increased sorting time with respect to the other Java configurations (about a 7% increase) is due to the overlapped I/O that occasionally requires the attention of the CPU and introduces some contention for the memory bus. The time spent blocking in the synchronization methods indicates that the CPU and disk workloads were not exactly balanced and that occasionally the CPU blocked while waiting for an I/O operation to complete.

# 6. LIMITATIONS AND FUTURE WORK

There are several limitations in the current prototype implementation that have yet to be addressed. Perhaps the most important remaining task is porting the AsyncFile library to platforms other than Solaris – the library should be easily portable to any OS providing the minimal asynchronous I/O primitives described in Section 2, but no real attempt has been made to do so thus far. An interim solution for Titanium implementations on platforms lacking kernel support for asynchronous I/O would be to implement the initiation routines with blocking semantics on those platforms. It may also be possible on some platforms to perform I/O thread creation and management entirely within the native routines to simulate the required behavior. Porting the AsyncFile library may be less of a problem for (non-Titanium) Java dialects that support arbitrary threading at the language level, because the library could be implemented completely in Java. However, bulk primitives would still be required at some level to achieve efficiency and the overheads of thread creation and management in Java might make a Java-level implementation unprofitable.

Another limitation of the current prototype implementation is that it only supports file I/O on single-dimensional arrays of non-reference types. The basic reason is that multidimensional arrays in Java are unstructured and their data elements are stored non-contiguously (multi-dimensional arrays are represented as a hierarchy of references to single-dimensional arrays which could possibly differ in size). In any case, a programmer could certainly perform I/O on the constituent single-dimensional fragments of a multi-dimensional Java array with the caveat that the application may have to store some additional application-dependent meta-information in order to recover the shape of a multi-dimensional array read in this fashion. It is not clear what it means to perform I/O on non-primitive (i.e. reference) types, although the object serialization approach pioneered in Java 1.1 is probably a good start. Titanium does not include serialization because it is not a Java 1.0 feature, however several studies have reported that standard object serialization is often too slow for data-intensive applications [4], [12], [16]. Welsh describes some interesting approaches to optimizing object serialization for file I/O [27].

The single-dimensional restriction is not a serious limitation in Titanium, because the language includes a more powerful structured array abstraction called grids that provide better support for multi-dimensional calculations. The bulk I/O extension described in this paper has also been successfully adapted to work with grids; the I/O performance gains are comparable, but the results were not presented here for brevity.

There are several open issues concerning the lack of strong file system consistency semantics in Java. The existing Java I/O libraries provide no OS-level file locking capabilities to limit concurrent accesses to a single file. Furthermore, they provide no strong guarantees about the behavior of concurrent accesses to the same region of a file when at least one thread is performing updates. AsyncFile does not currently address these deficiencies either, and strictly speaking, AsyncFile is just as safe as the existing libraries. However, since the interface does encourage concurrent, asynchronous file accesses, it might be advantageous to add some concurrency control features to help the programmer ensure correctness.

A final limitation (which is also shared by all the I/O libraries) is that nodes in a distributed system cannot use I/O objects created by other nodes. The Titanium memory model provides a global memory space that allows references to remote data objects, but

most distributed operating systems do not implement file descriptors that are portable across nodes, so the methods that perform kernel interactions would fail. This is not seen as a serious limitation in the presence of a network file system, because each processor can locally create its own file object that references the appropriate file.

Other areas for future work include providing some form of implicit data distribution – this is an important facet of parallel I/O optimization that is currently left up to the explicit control of the application. Another avenue for further exploration is adding support for implicitly asynchronous file operations – one could imagine a compiler optimization which converts blocking file operations to non-blocking ones (perhaps hoisting them to earlier locations in the program as well), and automatically inserts synchronization operations before the first use of a buffer after a non-blocking read. However, anecdotal evidence suggests that programs designed without thought given to such optimizations might not exhibit a large degree of CPU/disk parallelism without a very sophisticated compiler analysis and extensive optimizations.

## 7. CONCLUSIONS

This paper has presented two extensions to the Java I/O libraries that provide an efficient interface for high-performance applications that perform large amounts of file I/O. The bulk synchronous extensions add array operations to the existing I/O libraries, removing the bottlenecks associated with bulk I/O in the legacy interface. The new AsyncFile classes provides library support for asynchronous I/O, allowing applications to mask I/O latency with overlapped computation. Experimental results on the Titanium-based prototype demonstrate the extensions provide enormous performance gains exceeding 60x on a simple sorting benchmark. There is considerable room for future work, but it seems clear that these extensions are a valuable addition to the Java programmer's toolkit.

## 8. ACKNOWLEDGMENTS

This material is based in part on work supported by DARPA contract No. F30602-95-C-0136 and a Sloan fellowship. We would like to thank the entire Titanium team, especially Kathy Yelick and Ben Liblit for their invaluable help.

## 9. REFERENCES

[1] Acharya, A. et al., "Tuning the Performance of I/O Intensive Parallel Applications," Rice Center for Research on Parallel Computation (1995).

[2] Aiken, A. and Gay, D., "Memory Management with Explicit Regions," *Proc. of Programming Language Design and Implementation Conference*, Montreal (June 1998).

[3] Begel, A., "Titanium Threads," Technical Report (1998). Available online from [25].

[4] Carpenter, B. et al., "Object Serialization for Marshalling Data in a Java Interface to MPI," *Proc. of the ACM 1999 Java Grande Conference* (June 1999).

[5] Culler, D. et al., "Parallel Computing on the Berkeley NOW," *9th Joint Symposium on Parallel Processing* (1997).

[6] Dickens, P. and Thakur, R., "An Evaluation of Java's I/O Capabilities for High-Performance Computing," submitted to the ACM Java Grande 2000 Conference.

[7] Gay, D. and Aiken, A., "Barrier Inference," *Proc. of Principles of Programming Languages*, San Diego (January 1998).

[8] Gosling, J. and Steele, G., "The Java Language Specification," (June, 1996).

[9] Heydon, A. and Najork, M., "Performance Limitations of the Java Core Libraries," *Proc. of the ACM 1999 Java Grande Conference* (June 1999).

[10] Hilfinger, P., "Titanium Language Working Sketch, rev 0.22," (September, 1999).

[11] Jacobson, D. and Wilkes, J., "Disk scheduling algorithms based on rotational position," HP Laboratories Technical Report (1991).

[12] Judd, G., Clement, C. and Snell, Q., "Design Issues for Efficient Implementation of MPI in Java," *Proc. of the ACM 1999 Java Grande Conference* (June 1999).

[13] Kennedy, K., Koelbel, C. and Paleczny, M., "Scalable I/O for Out-of-Core Structures," Rice Center for Research on Parallel Computation (1994).

[14] Message Passing Interface (MPI) Standard 2.0 Specification, Chapter 9 (1997).

[15] Microsoft Windows 32-bit API, Microsoft Developer Network (1999).

[16] Nester, C., Philippsen, M. and Haumacher, B., "A More Efficient RMI for Java," *Proc. of the ACM 1999 Java Grande Conference* (June 1999).

[17] NOW Project web page. http://now.cs.berkeley.edu/

[18] Parallel I/O Archive. http://www.cs.dartmouth.edu/pario/

[19] Rosario, J., Bordawekar, R., and Choudhary, A., "Improved Parallel I/O via a Two-phase Run-time Access Strategy," *IPPS Parallel I/O Workshop* (1993).

[20] Rosario, J. and Choudhary, A., "High Performance I/O for Parallel Computers: Problems and Prospects," *IEEE Computer* (July 1993).

[21] Ruemmler, C. and Wilkes, J., "Disk Shuffling," Hewlett-Packard Software and Systems Laboratories Technical Report (1991).

[22] Ruemmler, C. and Wilkes, J., "An introduction to disk drive modeling," Hewlett-Packard Software and Systems Laboratories Technical Report (1994).

[23] Shriver, E., Merchant, A. and Wilkes, J., "An analytic behavior model for disk drives with readahead caches and request reordering," *Proc. of SIGMETRICS* (1998).

[24] Solaris 2.6 Reference Manual. http://docs.sun.com/

[25] Split-C Project web page. http://www.cs.berkeley.edu/Research/Projects/parallel/castle/split-c/

[26] Titanium Project web page. http://www.cs.berkeley.edu/Research/Projects/titanium/

[27] Welsh, M. and Culler, D., "Jaguar: Enabling Efficient Communication and I/O from Java," *Concurrency: Practice and Experience* (December, 1999).

[28] Worthington, B., Ganger, G. and Patt, Y., "Scheduling for Modern Disk Drives and Non-Random Workloads," U. of Michigan Tech Report (1994).

[29] Yelick, K. et al., "Titanium: A High-Performance Java Dialect," *ACM 1998 Workshop on Java for High-Performance Network Computing*, Stanford, CA (February 1998).